C++学习笔记总结8，9，10

2022年3月19日 21:00

- 关联容器
  - map类容器：map、multimap和unordered map（值对集合）
    - 实现方式
      - 相同
        - map和mutimap的容器是按照key值默认排好的，unordered map则不是
          - map和mutimap的实现方式为红黑树
      - 区别
        - unordered map实现方式为哈希表
    - 底层逻辑：
      - map、unorderedmap
        - 在这两类中均为key-value 一对一的关系。
      - multimap
        - 在multimap中关键词（key）是可以重复出现的
    - 对应的基本类型：pair
      - pair也是有一个key-value组成，与map不同，pair是一个，map是一组
      - 每一个map都由若干个pair组成，就像字符是字符数组的组成单位一样，pair是map的基本组成单位
      - pair的初始化方式
        - 值初始化
          - 我们可以使用make-pair<x,x>()来进行初始化 pair<char,int> b=make_pair<char,int>('a',2);
        - 直接初始化
          - 我们可以使用花括号或者圆括号对齐直接初始化
            - 花括号：pair<char,int> c('b',2);
            - 圆括号：pair<char,int> c{'b',2};
    - map类的使用（以map类为例，其他两个使用方法相同，特殊会注明）
      - 创建
        - 对于三类map都定义在与自己同名的头文件中
        - 创建map类的格式（以map为例子）
          - map<char,int> a;（空map）
          - 其中尖括号里面需要两个值分别代表key与value
      - 初始化方式
        - 直接初始化
          - 使用圆括号初始化：map<char,int> a（b）
          - 使用两个迭代器进行直接初始化：map<char,int>

a(x.begin(),x.end());

- 添加元素
  - 首先需要注意，向map类中添加的必须是pair类的元素
  - 函数类操作
    - insert（）（emplace（））以insert为例
      - 插入一个值： a.insert（k）k为pair，现在空的a中增加了一个新的元素
        - k可以的形式
          - {’ a'，2}//大括号类型
          - make-pair<char,int>('a',2);//makepair
          - pair<char,int>('a',2);//pair
        - 返回值类型
          - 对于map，返回一个pair类型，指向指定有key元素的map类迭代器和一个布尔类型
            - 如果插入成功：布尔类型返回1，迭代器指向成功插入的那个元素
            - 如果插入失败：布尔类型返回0，迭代器指向已经存在的元素
          - 对于multimap：只返回指向插入元素的迭代器
      - 利用迭代器插入：a.insert(iterator x,iterator y),插入【x，y）之间的元素
        - 插入之后不返回元素
  - 下标类操作
    - 对于map和unordered-map：
      - 支持下标类操作，与数组类似
        - a【k】----》返回map a中关键字（key）为k的value，若没有则创建并且将value进行值初始化
    - 对于multimap
      - 不支持下标操作，因为其有多个key
- 查找元素
  - 查找元素在哪里
    - 函数类查找
      - find（k）：单一元素查找，如果在map里返回迭代器，不在返回尾后迭代器
    - 下标类查找
      - c.at(k):可以直接查看下标为k的元素，如果在就返回元素，不在返回一个异常，只能在map和unorderedmap上使用
  - 查找元素的数量
    - count（）：查看key元素对应的数量，对于不允许有多个key的只会返回0和1

        - 删除元素
            - erase（k）：删除关键词k所指向的元素
                - 对于map成功返回1，失败返回0
                - 对于multimap成功返回删除的数字，失败返回0
            - erase（x）：删除迭代器x所指向的元素
            - erase（x，y）：删除迭代器【x，y）之间的 元素
    - set类
        - set类容器：set，multiset，unordered-set
            - set：元素的集合，每个元素只能出现一次
            - multiset：每一个元素可以出现多次
            - unordered-set：相当于一个没有排序的map
        - set的初始化
            - set初始化与map不同，set只有一个参考元素，其关键字和值是一体的但是其初始化方式与map完全相同
        - set其他的相同操作均与map相同
- 内存分配
    - 使用智能指针分配内存
        - 智能指针的分类
            - share-ptr：智能指针，允许多个指针指向一个对象
            - unique-ptr：智能指针，一对一类型（只能指向一个对象）
            - weak-ptr：是一个弱引用，指向指针所指向的对象（但是不会增加share-ptr的计数）
        - 智能指针的特性
            - 定义在memory中
            - 智能指针在出定义域的时且引用为0的时候，会调用析构的操作（析构函数等）自动释放指针所指向内存的空间
        - 智能指针的创建和初始化
            - 我们使用make-share来创建智能指针，其他两个也是如此。
              share-ptr<数据类型> a=make-share<数据类型>（值初始化）
        - 智能指针（引用）的添加
            - 拷贝时候添加
                - 指针进行值传递
                - return指针
                - 等于（拷贝构造函数导致指针相等）
        - 智能指针的删除
            - 出作用域后调用析构函数减少计数/释放空间
    - 使用new普通指针分配内存
        - 普通指针的特性
            - 分配内存，一旦分配必须手动释放
        - 普通指针的创建

int *p=new int（初始化）

- 普通指针的删除
  - 我们使用delete去删除一个指针指向的内存
  - 对于类会调用析构函数释放类所占用的空间

- 普通指针和智能指针之间的关系
  - 智能指针可以使用get（）函数从智能指针中拿出对应的指针，但是这很危险
  - 普通指针可以通过智能指针创建的时候将普通指针直接初始化给智能指针，但是这可能会造成空悬指针，故需要及时将被智能指针托管的普通指针指向nullptr；
- 使用alllocator分配内存（不常用）
  - allocator类定义在头文件memory中，它可以将内存分配和对象构造分来，即先分配内存，再构造对象
  - allocator的创建
    - 我们创建一个allocator，提前告知他要分配什么样类型的内存，但是不告诉什么时候开始分配和它需要分配多少，也不装入对象
      allocator<int> a;
  - allocator分配内存
    - 我们通过函数allocate()告诉allocator现在需要分配内存和分配多少，但是我不装入对象
      - a.allocate()分配一段原始未构造的内存，指出我要分配多少对象
      - allocate返回一个指针，指向分配内存的最开始部分
  - allocator构造对象（已经在C++ 20中删除）
    - 我们使用construct（）告诉allocator我要构造对象了
      - a.construct(p，args)构造对象
        - p为调用allocate函数返回的指针。p调用的时候你可以对指针进行移动（p++等等）
        - args时对应之前创建类型的构造函数，args可能有一项，也可能有两项
  - allocator删除元素（已经在C++ 20中删除）
    - 对于已经构造了对象的元素，我们可以使用destroy销毁元素
      - a.destroy(--q);q为末指针
  - allocator释放内存
    - 对于已经销毁的元素，可以调用construct重新创建内u你，也可以调用deallocate彻底删除它
    - a.deallocate(p,n)
      - p：指向调用最开始的那个指针
      - n：之前创建的数目

- 类的拷贝、控制与移动
  - 基本构造函数
    - 构造函数

- 操作
  - 其中，“：”后面可以直接赋值对象，大括号内是一个作用域，可以进行内存直接分配，流的返回等等

```
HasPtr::HasPtr(const string a,int i):i(i)
{
    ps=new string(a);
    pt=new size_t(1);
}
```

- 被调用的时机：在一个对象创建的时候进行调用

```
HasPtr a(s,2);
```

- 默认构造函数：HasPtr（）=default

- 拷贝类函数
  - 拷贝构造函数
    - 操作
      - 与构造函数不同的地方是：仅含有一个类的引用

```
HasPtr::HasPtr(const HasPtr& a)
{
    //操作
    ps=a.ps;
    i=a.i;
    pt=a.pt;
    ++*pt;
}
```

    - 被调用的时机
      - 作为参数传递给函数的时候（或者隐式转换的时候（前提是非explicit的））

```
int x(HasPtr a)
```

      - return对应类的时候

```
HasPtr x(HasPtr a)
{
    cout<<"xx";
    return a;
}
```

      - 对一个新创建的变量直接进行拷贝的时候

```
HasPtr b=a;
```

    - 默认构造函数：当类没有定义拷贝构造函数的时候，编译器会自动生成一个构造函数
  - 拷贝赋值运算符
    - 操作
      - 我们一般使用operator关键词+返回一个引用作为拷贝复制运算符

```cpp
HasPtr& HasPtr::operator= (const HasPtr& a)
{
    //操作
    this->~HasPtr();
    ps=a.ps;
    i=a.i;
    *pt=++*(a.pt);
    return *this;
}
```

- 被调用的时机
  - 在两个已经创建完成的变量之间使用

```cpp
    string s("hello");
    HasPtr a(s,2);
    HasPtr b("bye",3);
    b=a;
```

- 移动类函数
  - 右值体系
    - 右值引用
      - 右值满足的特点
        - 形式上：右值往往是那种字面值，如5，8， “hello”字面值
        - 内存上：右值一般不占内存（没有地址）
      - 定义：我们用&&来对定义右值引用
        - int &&i=2;
      - 右值引用的特点
        - 右值引用得到的变量是个左值
          - 如int &&i=2，i就是一个左值
        - 右值引用只能绑定到临时对象上
          - 返回非引用类型的函数是右值，可以绑定
          - 返回数字类型的函数是右值，可以绑定
        - 右值引用延长了右值的生存周期，从立刻被销毁到作用域结束后才被销毁，因为它变成了一个左值却具有右值的特性
          - 如int &&i=2，2本来是一个字面值，如果是左侧是引用则结束赋值后就会销毁，但是如果是右值它依然保持着
      - 满足使用右值引用的条件
        - 临时对象
        - 对象即将被销毁
        - 该对象没有其他用户
        - 使用右值引用可以自由接管所引用对象的成员
        - 如果移后源对象的不确定状态，不对其调用std：：move（）即更改他的状态
      - std：：move（）
        - std：：move定义在utility函数

- ◇ std：：move（a）作用
  - ▸ 简单说就是将所有权进行转移，若转移成功后，所有权到达新的对象，原对象不再拥有所有权，再次访问会发生未知错误，例如段错误，未知值等等
    - – 对于如果只是调用std::move然后使用右值引用接收后，原对象仍然拥有该内存的所有权，因为只是获取了右值引用，并未完全移动给另一个对象

```cpp
#include <iostream>
#include <utility>
using namespace std;
int main()

{
    int i=2;
    int &&c=std::move(i);
    int &e=i;//此时的i依然是掌控着“2”这个数值，c只是获取的它的右值引用
    const int &d=c;
    cout<<e<<c;
}
```

    - – 如果该右值是以匿名对象的方式传入，则原对象的所有权丢失，再次调用将会发生错误

```cpp
string A("abc");
string C(std::move(A));                 // 和上面不一样的是
cout << A << endl;                      // output ""

cout << C << endl;                      // output "abc",
```

- ◇ std：：move（a）一旦这个表达式运行，我们将只能对a进行销毁和赋值，最好不要考虑让移动后的a干其他事情
- ◇ 注意不要把右值和move混为一谈，右值是右值，move是更换交接权力！

- ◆ 移动迭代器
  - ◇ 移动迭代器解引用能生成一个右值引用
  - ◇ 我们可以调用make-move-iterator来将一个普通迭代器转化为一个移动迭代器
  - ◇ 可以将一个移动迭代器传递给uninitialized-copy

- ▪ 移动赋值运算符
  - □ 操作
    - ◆ 移动拷贝参数的参数是一个右值引用的参数，传入之后已经被move的x将只能被赋值或者等待销毁，在程序的最后，传入的参数会被析构

```
xxx& xxx::operator=(xxx&& x)
{
    a=x.a;
    b=x.b;
    p=x.p;
    x.p=nullptr;
    cout<<"move=";
    return *this;
}
```

- 被调用的时机
  - 显示用字面值初始化的时候进行调用

```
dx=std::move(cx);
xxx ex=std::move(dx);
```

- 默认构造函数
  - 定义了一个以上的拷贝操作，就默认不会定义移动构造函数

- 移动构造函数
  - 操作
    - 移动拷贝参数的参数是一个右值引用的参数，传入之后已经被move的x将只能被赋值或者等待销毁，在程序的最后，传入的参数会被析构

```
xxx::xxx(xxx&& x)noexcept
{
    a=x.a;
    b=x.b;
    p=x.p;
    x.p=nullptr;
    cout<<"move";
}
```

  - 被调用的时机
    - 传入了一个的右值

```
xxx ax(a,b,p);
xxx bx(c,d,q);
auto cx=ax;
auto dx=bx;
dx=std::move(cx);
```

  - 默认构造函数
    - 定义了一个以上的拷贝操作，就默认不会定义移动构造函数

- 析构函数
  - 操作
    - 析构函数由一个“~”和对应类名组成

```
HasPtr::~HasPtr()
{
    --*pt;
    if(*pt==0)
    {
        delete pt;
        delete ps;
    }

}
```

- 被调用的时机
  - 可以直接作为对象的函数进行调用

```
HasPtr& HasPtr::operator= (const HasPtr& a)
{
    //操作
    this->~HasPtr();
```

  - 在一个函数结束的时候进行调用，释放之前由拷贝构造函数合成的函数

```
HasPtr x(HasPtr a)
{
    cout<<"xx";
    return a;
}
```

    调用析构函数，释放a
  - 在主函数结束的时候按逆序释放所有由构造函数，拷贝构造函数合成的函数
    - 例如下图，释放的顺序依次是xx，b，a

```
    {
        string s("hello");
        HasPtr a(s,2);
        HasPtr b("bye",3);
        b=a;
        auto xx=x(a);
```

- 默认构造函数：在没有定义析构函数的时候，编译器会自动合成一个
- 特殊提醒
  - 析构函数是不能重载的函数，因为他没有任何的参数
  - 一般不能delete析构函数
  - 析构函数本身不能直接销毁成员，它可以销毁直接分配出来的空间
    - 例如如果假设HasPtr则在调用析构函数的时候，只会销毁手动分配的空间，但是其他的成员及其对应的值是不会被销毁。

```
HasPtr& HasPtr::operator= (const HasPtr& a)
{
    //操作
    this->~HasPtr();
    ps=a.ps;
    i=a.i;
    *pt=++*(a.pt);
    return *this;
}
```

        - 成员是析构函数体之后隐含的析构阶段中被销毁的。在整个对象销毁过程中，析构函数体是作为成员销毁步骤之外的另一部分而进行的

- 五大特殊的类函数之间的相同和不同
    - 对于构造函数，拷贝构造，移动构造，都可以使用成员初始化器（：后面的初始化）进行初始值的获取
    - 对于每一个类中定义的函数，可以使用noexpect通知函数不应该抛出异常，在移动构造函数最为常用
    - 与拷贝操作不同，移动操作永远不会隐式定义为删除的函数
    - 如果我们显式要求编译器生成=default的移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数